Graphics Library

Profiling & Optimizations

1. Introduction

a. Graphics Library

The GigaVoxels-GigaSpace engine is based on OpenGL. As OprenGL API has deprecated the traditional fixed fonction pipeline of OpenGL 2.0, code should be based on OpenGL 4.0 as much as possible (ex : all matrices support has disappeared, and GLSL shaders are mandatory). Compatibility mode still enable to work with all mixed API. Recent OpenGL 4.x features must be used carefully by checking user device hardware at initiatlization. For instance :
- 4.2 : image load/store that enable user to write to texture (used by the GigaVoxels GLSL renderer)
- 4.3 : GLSL compute shader (this bypasses the traditional GLSL pipeline)
This leads to talk about extensions, because OpenGL ABI has been fixed since 1.1 ou 1.2 (<= check this)
Note : the library could be extended to work with DirectX.

Extensions
We use the GLEW library to load and check graphics library extensions at initialization.
Note : On Windows, it seems that GLEW as problems to handle OpenGL 4.3 features. Some features need to been loaded manually with the help of “function pointers” and “#define” enumerations.

GLM
As OpenGL 4.x API as deprecated matrices suuport, the GLM mathematic library is used to handle vectors and matrices and their associated operations.
The idea is to not pollute to much the GigaVoxels-GigaSpace library. For instance, try to use “float*” arguments in function parameters, then build glm::mat4 inside in order to stay generic.

b. Window Management

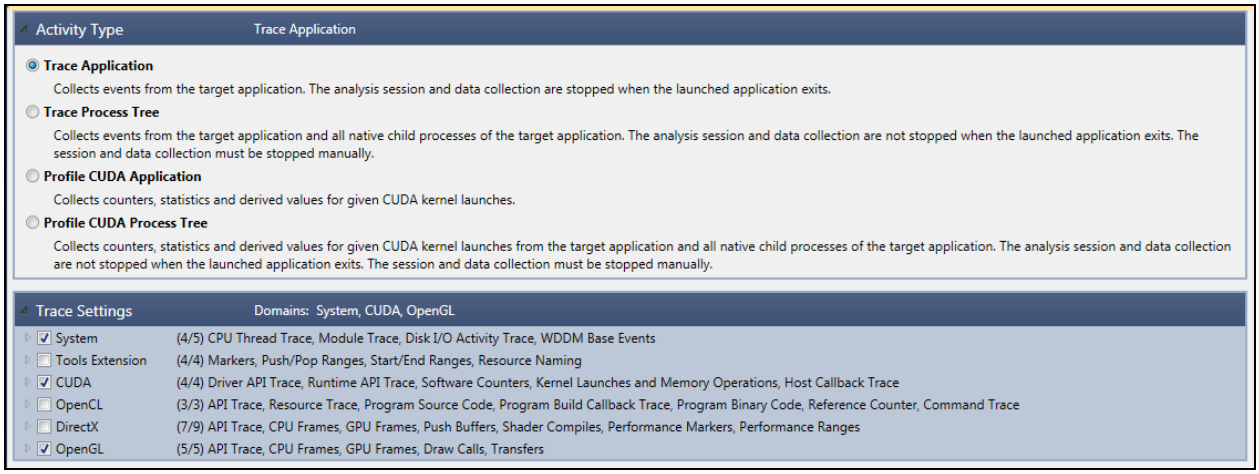
Window management is based on the Qt framework.
Actually based on 4.7 or 4.8 version, we should pass to 5.x for the Release.

QGLViewer library is then used to handle OpenGL easily. It is written in OpenGL 2.0 fixed fonction pipeline spirit. OpenGL 4.x is not in the QGLViewer team’s roadmap.

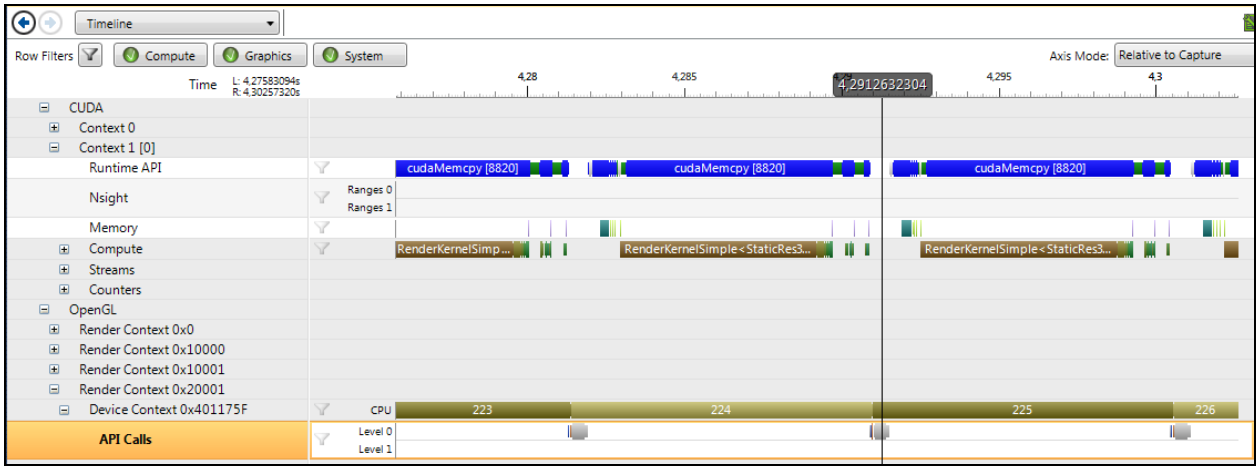
2. Profiling

a. Configuring the NSight Environment

To be able to profile OpenGL calls, activate the “OpenGL Trace Settings”.
Expand the OpenGL item and choose all features you want to trace (vertex array, texture, etc...).



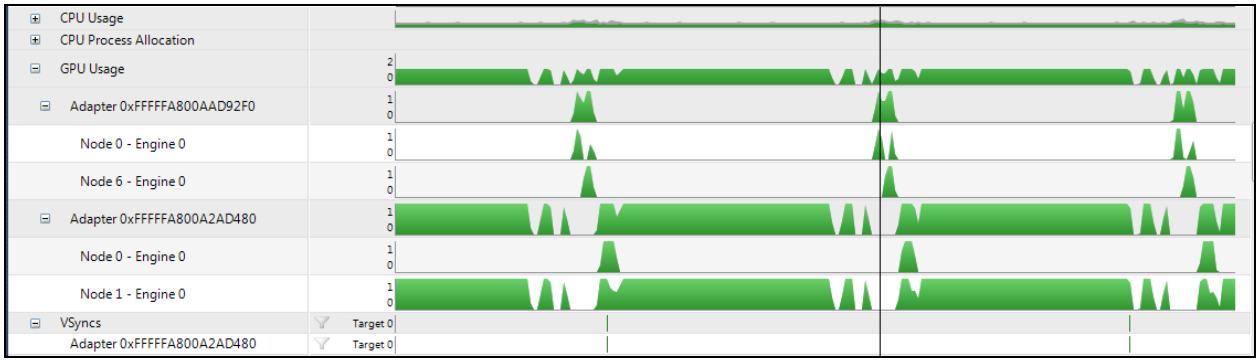
An example of trace with NSight timeline.
In the OpenGL item, expand the Render Context and Device Context items to be able to see all OpenGL API calls.
In the following example, NSight shows all frames (check at “Device Context” : here we have zoomed on frames 223, 224, 225)



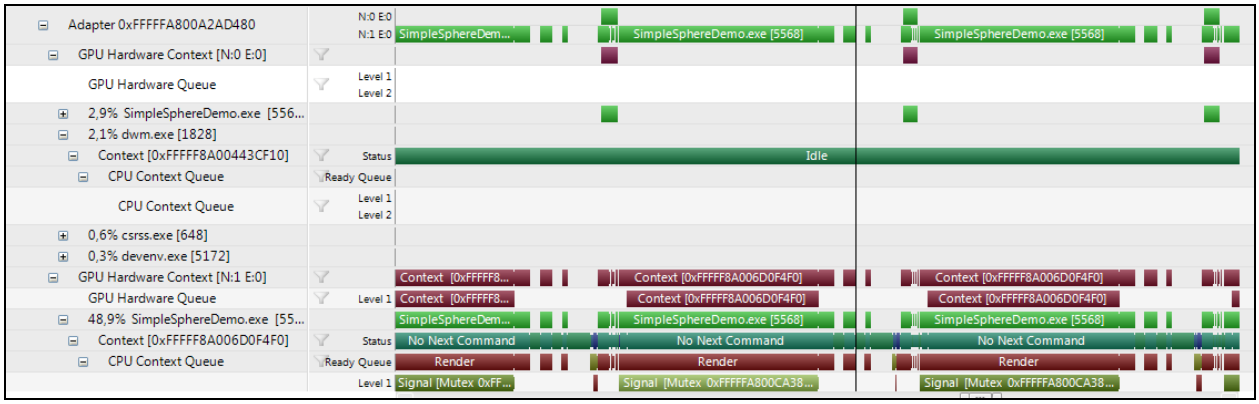
We are now able to identify on the same graph, the start/end of each GL function call as well as the CUDA ones in the “Compute” item.

For instance, we can locate “RenderKernelSimple”, the main GigaVoxels renderer CUDA kernel, inside each frame call.

NSight can also show GPU usage :



And low level activity :



b. Example : Simple Sphere demo

Here’s the timeline of the GL calls during 1 frame :

Inheriting from the QGLWidget class, the main QGLViewer class function is :
virtual void paintGL();

For rendering, QGLViewer gives access to 5 functions called each frame :

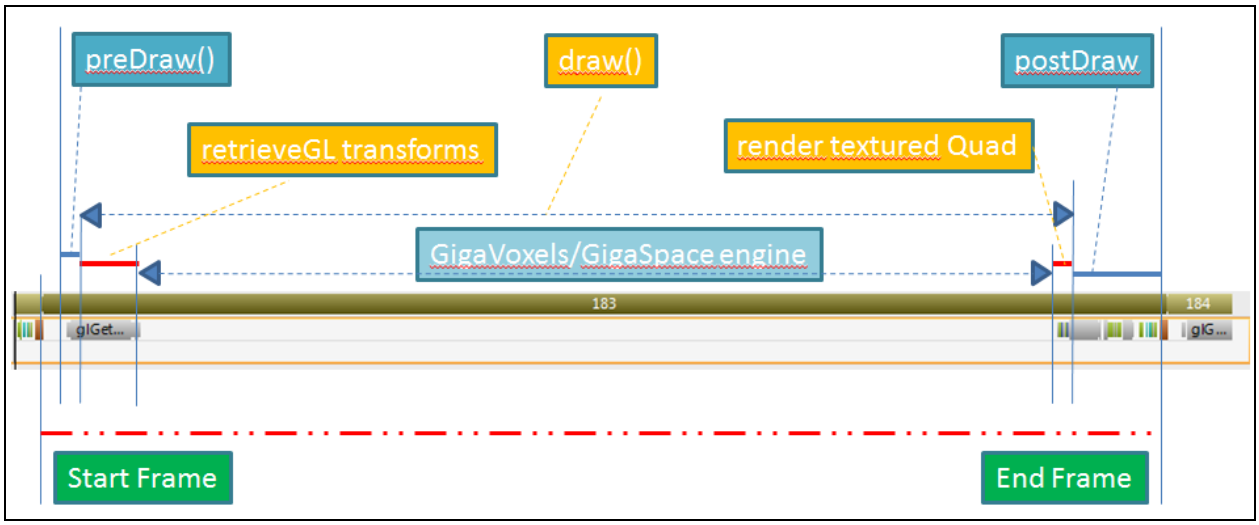
virtual void paintGL();
virtual void preDraw();
virtual void fastDraw();
virtual void draw();
virtual void postDraw();

```
void QGLViewer::paintGL()
{
    if (displaysInStereo())
    {
        for (int view=1; view>=0; --view)
        {
            // Clears screen, set model view matrix with shifted matrix for ith buffer
            preDrawStereo(view);
            // Used defined method. Default is empty
            if (camera()->frame()->isManipulated())
                fastDraw();
            else
                draw();
            postDraw();
        }
    }
    else
    {
        // Clears screen, set model view matrix...
        preDraw();
        // Used defined method. Default calls draw()
        if (camera()->frame()->isManipulated())
            fastDraw();
        else
            draw();
        // Add visual hints: axis, camera, grid...
        postDraw();
    }
    Q_EMIT drawFinished(true);
}
```

- preDraw() : used to the FrameBuffer (color | depth)
- fastDraw() : by default, it calls draw()
- draw () : the main user-defined function to render. GigaSpace is launched here
- postDraw() : used to display visual hints like axis, grid, text, etc...

Here is 1 frame zoom :

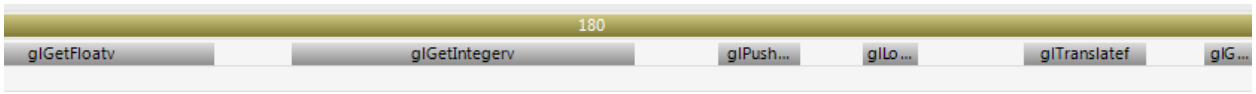
There 3 parts : start frame, render frame and end of frame :



preDraw()
Here is the QGLViewer preDraw() : clear framebuffer, retrieve/set projection and modelview matrices (glLoadMatrix, glMatrixMode, glMultMatrix...).

draw()
Then, the user-defined draw().
The default mode of the GigaSpace engine uses a render to texture mode with the help of the CUDA Graphics Library Interoperability. It starts by binding a local framebuffer with a texture bound to color_attachement 0.

Then the engine needs to retrieve all the GL matrices for modeling, modelview, projection an viewport transfom, so we see a succession of call to OpenGL context with glGetFloatv, glGetIntv...



This step of retrieving info from the GL context seems to be slow, or may stall the cpu/gpu ? It should be better to track a stack of matrices and only ask GL context if requested. The use of the GLM mathematic library, or internal matrices of QGLViewer, could speed this part of code.
The GigaSpace engine is then executed to render in the OpenGL texture. Finally, the texture needs to be display a full-screen quad. Here, we use OpenGL Immediate Mode rendering with deprecated OpenGL 2.0 fixed function pipeline. With many call OpenGL functions, this generates overhead to the graphics driver. It should be better to use a Vertex array, VBO, plus a GLSL programmable shader.

postDraw() to add visual hints
NSight shows that the postDraw() methods calls a lot of OpenGL API functions, modifying the OpenGL state machine, while saving/restoring all OpenGL context with the help of :

```
// Save OpenGL state
glPushAttrib( GL_ALL_ATTRIB_BITS );
// Restore GL state
glPopAttrib();
```

To optimize this, it could be better to call the postDraw() method only if we need it :
- show axis, show grid, etc… based on user-defined boolean variables

3. Optimizations

a. QGLViewer

b. OpenGL

c. CUDA

Algorithm, modification : don’t fetch/write PBO or texture in CUDA kernel if not required.

d. CUDA / GL interoperability

Instead of using PBO (Pixel Buffer Object) that CUDA see as a (void*), use direct access to texture or and/renderBuffer objects, seen as cudaArrays that can be read/write with the help of texture and or surfaces.
However, CUDA can’t handle DEPTH texture. So we’re still forced to use PBO and intermediate copy, that slow down application.

4. Design & Architecture

The GigaSpace engine has been divided in three shared libraries :
- *GsGraphics* : to manage core OpenGL features (shaders, hardware device management/configuration, debugging, etc...). The idea behind is to always think how to be able to change from OpenGL to DirectX library.
- *GsCompute* : to manage core CUDA features (hardware device management, error management, etc...). The idea behind is to always think how to be able to change from CUDA to OpenCL library.
- and the main *GigaSpace* shared library.

IMPORTANT
Beware, CUDA has not yet provided a linker for GPU code in shared libraries. It only work for static library. So all code must be in the same compilation unit.

GsGraphics and GsCompute have been designed to work stand-alone in other applications (ex : the GigaSpace viewer). All OpenGL and CUDA code should be placed here in priority taken into account the previous warning about the lack of a device linker.

References

OpenGL
- “OpenGL Insights” book
- “Approaching Zero Driver Overhead”, GDC 2014 (Game Developers Conference), Cass Everitt (NVidia) - John McDonald (NVidia) Graham Sellers (AMD) - Tim Foley (Intel)
- “Low-level Thinking in High-level Shading Languages”, GDC 2013 (Game Developers Conference), Emil Personn - Head of Research, Avalanche Studios
- “Low-level Shader Optimization for Next-Gen and DX11”, GDC 2014 (Game Developers Conference), Emil Personn - Head of Research, Avalanche Studios
- “Modern OpenGL Usage: Using Vertex Buffer Objects Well”, Mark J.Kilgard (NVidia Corporation), Austin - Texas, 2008

WDDM
- Wikipedia : http://en.wikipedia.org/wiki/Windows_Display_Driver_Model
- Windows Display Driver Model (WDDM) : Microsoft MSDN => http://msdn.microsoft.com

CUDA
- ...